University of Florida

Computer and Information Science and Engineering

A Visualization and Explanation Tool for Debugging ECA Rules in Active Database

S. Chakravarthy
Z. Tamizuddin
J. Zhou

EMAIL: sharma@cis.ufl.edu

WWW: http://www.cis.ufl.edu/~sharma

Tech. Report UF-CIS-TR-95-028

November 1995

(Appeared in RIDS '95 Int'l Workshop)

Computer and Information Science and Engineering Department
E301 Computer Science and Engineering Building
University of Florida, PO Box 116120
Gainesville, Florida 32611-6120

Table of Contents

A Visualization and Explanation Tool for Debugging ECA Rules in Active Databases *

S. Chakravarthy **Z. Tamizuddin** **J. Zhou**

Database Systems Research and Development Center
Computer and Information Science and Engineering Department
University of Florida, Gainesville, FL 32611
email: sharma@cis.ufl.edu

Abstract. Using ECA rules in active database systems for real-life applications involves implementing, debugging, and maintaining large numbers of rules. Experience in developing large production rule systems has amply demonstrated the need for understanding the behavior of rules especially when their execution is non-deterministic. Availability of rules in active database systems and their semantics creates additional complexity for both modeling and verifying the correctness of such systems. As part of Sentinel – an Object-Oriented Active DBMS, we have developed a visualization tool to help understand the behavior of rules defined as part of an active database application. This is especially important in active databases as rules are invoked (as a side effect) based on event occurrences (both primitive and composite) and are executed concurrently based on user-provided priority information. In this paper, we describe the rationale for the development of the tool, how it has been implemented exploiting the architecture of Sentinel, functionality of the resulting tool, and show several screen dumps to provide a feel for the information presented by the visualization tool.

1 Introduction

Using ECA rules in active database systems for real-life applications involves implementing, debugging, and maintaining large numbers of rules. Experience in developing large production rule systems has amply demonstrated the need for understanding the behavior of rules especially when their execution is non-deterministic. Availability of rules in active database systems and their semantics creates additional complexity for both modeling and verifying the correctness of such systems.

For the effective deployment of active database systems, there is a clear need for providing a debugging and explanation facility to understand the interaction – among rules, between rules and events, and between rules and database objects.

* This work is supported by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.

Due to the event-based nature of active database systems, special attention has to be paid for making the context of rule execution explicit [DJP93]. Unlike an imperative programming environment, rules are triggered in the context of the transaction execution and hence both the order and the rules triggered vary from one transaction/application to another. Use of the nested transaction model for rule execution in Sentinel [Bad93, Tam94] provides such a context. Our graphics visualization tool (which uses Motif) displays transaction and rule execution by using directed graphs to indicate both the context (i.e., transactions/composite events) and the execution of (cascading) rules. The same mechanism will be used for run-time as well as post-execution visualization of event definition, detection, and rule execution. In both modes, we provide interactive facilities (e.g., click on a rule to display the objects used, chain of rules it is part of, rules that it has signaled) for obtaining a detailed understanding of rule execution.

The model of the traditional debuggers (of conventional programming languages) is not adequate for debugging rules in active database systems. The emphasis in traditional debuggers is on low level details such as program variables, subroutines, pointer referencing/dereferencing. The rationale behind the rule debugger is to help understand the interaction – among rules, between rules and events as well as between rules and the database objects (including the objects held by a transaction/subtransaction). This is in contrast with the conventional debugging of programs. The interaction among rules refers to such details as nested triggering of rules, the events which cause the rule to fire or the context in which the rules are fired. The rules also affect the state of the database by way of modifying database objects and as a consequence acquiring locks. The rule debugger shows the rule-database interaction in terms of the locks acquired/released and the transactions in which the rules fire. The debugger is not meant to replace a conventional debugging tool (e.g., xdb), but to provide a tool at a higher level of abstraction to cater to ECA rule design, implementation, testing, and debugging.

The rest of the paper is structured as follows. Section 2 discusses the Sentinel architecture to briefly describe rule execution context and the functional modules introduced to make Open OODB active. Section 3 elaborates on the design of the visualization and explanation tool. Section 4 describes the implementation aspects. In section 5 we show several screen dumps produced by the tool at different stages of a sample application execution. Section 6 contains conclusions.

2 Sentinel Architecture

In this section, we briefly discuss the overall architecture of Sentinel [CM94, AMC93, Bad93, CKAK94, CKTB95, Tam94] and its functional modules. The Sentinel architecture shown in Figure 1 extends the passive Open OODB system [WBT92, OOD93]. The Open OODB toolkit uses Exodus as the storage manager and supports persistence of C++ objects. Concurrency control and recovery for the top level transactions are provided by the Exodus storage manager. Nested

transactions are supported in the client address space and a separate lock table is maintained by Sentinel. This essentially gives a two level transaction management (top level transaction concurrency is provided by Exodus at the server and the nested transaction concurrency is provided by Sentinel for each client). There is no recovery at the nested subtransaction level. The Sentinel architecture allows the execution of subtransactions that are *not* necessarily rules. A full C++ pre-processor is used for extending the user class definitions as well as application code. To make Open OODB active, the following extensions were made.

- ECA rules can be incorporated either as part of the class definition or as part of the application code. This allows the specification of class level and instance level definition of rules and events. Additional (Open OODB has its own pre-processor) Sentinel pre-processors were written to preprocess and translate the event and rule definitions into appropriate C++ code for event detection and rule execution.
- Detection of primitive events [CKAK94] was incorporated by adding Notify (a method call to the event detector class) into the wrapper method of the Open OODB. The wrapper method permits us to invoke a notification when an event occurs and conveys it to the composite event detector.
- To detect composite events a composite event detector [CM94, CKAK94] has been implemented. Each Open OODB application has a composite event detector. The event detector is implemented as a class and we have a single instance of this class per application. This is shown in Figure 1 as the local composite event detector. Also there is a clean separation between event detection and application execution.
- To support rule execution, we have extended the skeletal transaction manager of the Open OODB to a full-fledged transaction manager supporting the nested transaction model. The design of this transaction manager was described in [Bad93] and implemented on a previous version of Open OODB (Zeitgeist). The nested transaction manager has been redesigned and implemented for Sentinel [Tam94]. It is beyond the scope of this paper to describe the implementation of the nested transaction manager.
- As shown in the Sentinel architecture we have designed and implemented a Rule visualization/debugger module to provide feedback about event detection and rule execution. We discuss the design and implementation of the visualization tool in the next section.

3 Design of the Visualization and Explanation Tool

In a conventional imperative program debugging environment, the factors considered are variables, subroutine calls, exceptions (stack overflows), pointer referencing/dereferencing etc. The user knows the context in which to debug the program. For example, the instructions of a program are executed in a fixed

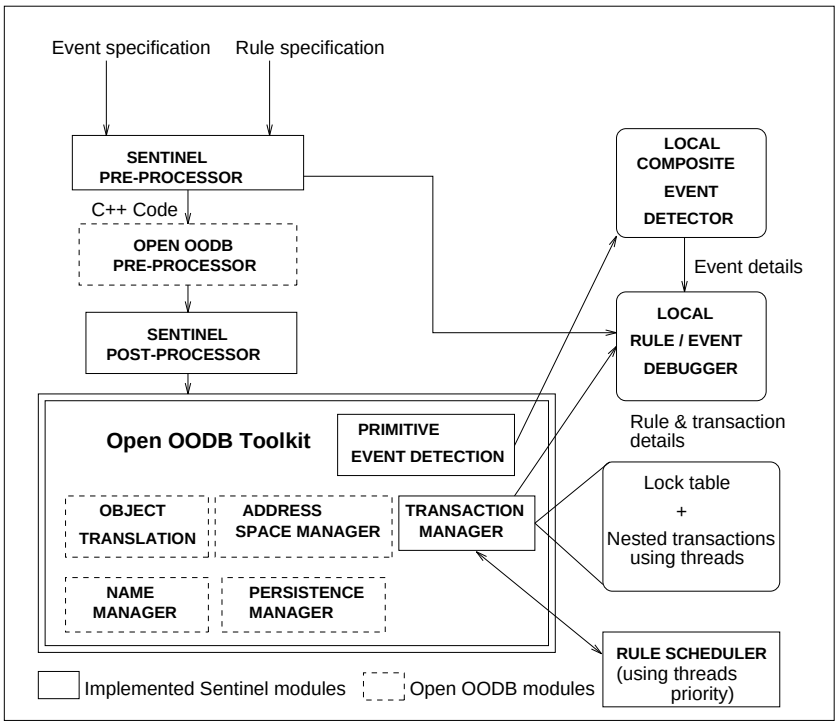

Fig. 1. Sentinel Architecture

sequence given by the programmer. If there is an error such as an overflow, non-existent pointer dereferencing etc, then the program fails and with the help of the debugger the user can locate where the error has occurred. The debugger aids the user in this process by furnishing low-level details, such as the line number of the program where the error occurred, variables accessed.

When we consider debugging in the context of an active database system, we need to take into account several factors:

- **Database component:** The database operations are performed on database objects. The database operations are carried out in well defined atomic units called transactions. In order to ensure atomicity and the correctness of the operations, locking is used according to the concurrency control mechanism used. Hence to get a complete picture of what is happening in a database system we need to take into account transactions, database objects and locks held (especially when there are concurrent transactions).
- **Rule and event interaction:** The active functionality of a database management system adds another dimension to the process of debugging. We have to consider the interactions among rules, between rules and events and between rules and database objects. When event(s) are raised appropriate

rule(s) are triggered. A rule may raise an event which may cause some other rule to fire. Hence we may have a nested execution of rules. An event may trigger several rules simultaneously. The rule and database objects interaction is in terms of locks acquired/released on objects in the process of rule execution.

We used the following rationale for obtaining the features of a rule debugger for an active object-oriented database management system:

- The rule debugger should concentrate on the high level details such as rule-event interaction, interaction among rules, and interaction of rules with database objects, rather than the low-level details typically provided in a programming environment.
- The debugger should show the execution of rules graphically preserving the triggering order, current status, and other relevant details. This is particularly useful when there is a nested execution of rules.
- In addition to the rule trace, the context (i.e., the events raised, whether the event was raised from within a rule or the top level transaction) should be shown.
- The debugger should allow the user to monitor specific rules/events of interest. This is useful in applications having a large number of rules.
- Finally, it should be possible to visualize application execution either at run-time or after the execution of an application for the analysis of rule execution. This should be accomplished without having to change either the architecture of Sentinel or the visualization tool.

The architecture for the visualization tool is dictated by the architecture for event detection in Sentinel. The functional architecture of the Visualization tool is as shown in the Figure 2.

As illustrated in Figure 1, the visualization tool communicates with the local event detector and the rule manager of each application. Since the local event detector detects the occurrence of an event and notifies the appropriate rule to be fired and is aware of the event object and the rule object ids, the runtime information about the event occurrence and rule execution is obtained from the local event detector. From the rule manager we get the transaction ids of the subtransactions within which the rules are executed.

We have designed and implemented the visualization tool to monitor rules and events within an application. We have adopted a *problem oriented* and *coupled approach* to debugging. Since we have adopted a coupled approach, the debugger and the active database system need to communicate. The way communication is achieved *for post-execution debugging* is via log files. The log files contain information regarding rule/event definitions, rule/event object id's, the actual occurrence and firings of events and rules, respectively, and other transaction related information. From the rule/event definition we can infer which rule

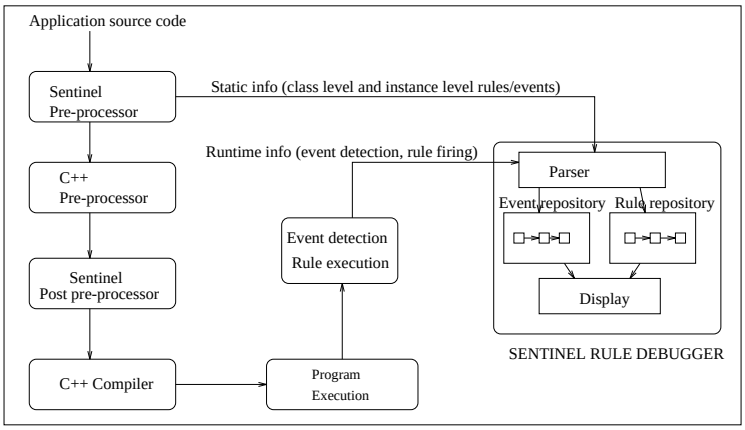

Fig. 2. Functional Modules

subscribes to which event and hence rule-event interactions. The transaction related information gives the rule-database interactions. The information in the log files is obtained by the pre-processor, event detector, and the rule manager. The visualization tool then reads these files to obtain the trace of event occurrences and rule execution. It is important to note that, currently, there is no runtime visualization done by the visualization tool. We say that runtime visualization is achieved when the visualization tool shows the occurrence of events and rule firings as and when they happen. If this were to be achieved then the communication between the visualization tool and the database system has to be done in real-time using sockets or named pipes. However, currently the visualization tool gathers information from the log files already created by the application execution and simulates runtime visualization. It makes only a single pass on the log files to obtain the relevant information and build in-memory data structures used by the visualization tool.

The input to the debugger consists of:

- The class level and the instance level rule and event definitions: This information is supplied by the user in the application program using the event [CM94] and rule definition [CKAK94] language. Since we have incorporated the event and rule definition language as extensions of C++ language, we preprocess the definitions. During this process, the preprocessor gathers this static information in the form of a file to the visualization tool.
- The event detection information: This is the runtime information obtained on the occurrence of events and the creation of event objects. This information is obtained at two points in the execution of the application program. First, when the event objects are created, the event name and the event object oid is given to the rule debugger. This information is used later by the rule debugger to associate the event object oids with their user supplied names.

As mentioned previously the object id information is particularly useful in the case of composite events. Finally when the event is actually raised it is notified to the rule debugger by the event detector.

- Finally, the rule firing information is furnished to the visualization tool in the same way as the event detection information. The rule object identifiers and the user supplied rule names are associated when the rule objects are created. In addition to the rule firing information, the transaction in which rules where fired, the locks acquired/released on database objects are also furnished. This helps the rule debugger to associate the rules in the transactions in which they were fired and also the objects which were accessed in the process of rule execution.

All the information mentioned above is furnished to the visualization tool in the form of two files. One of the files contains the static information on class and instance level events and rules. The second file contains the runtime information on event and rule object oids, the occurrence and firing of rules and the transaction and object access information.

4 Implementation of the Visualization Tool

We have implemented a visualization tool for active DBMS in an object-oriented context. The rule manager supports the event-condition-action paradigm as mentioned before. Apart from tracing the execution of rules, the visualization tool also keeps track of the events (both primitive and composite). As mentioned in [DJP93], the tracing of events gives important hints to the user. The event-rule cycle allows the user to not only to know which rules are fired but also which event(s) caused the rule(s) to fire. The occurrence of the events sets the context for the rule execution.

As mentioned earlier the input to the visualization tool is in the form of two files. One of the files furnishes static information in terms of the definition of events and rules, which rule subscribes to which event. This file is in fact generated by the preprocessor of Sentinel. The second file is generated at runtime and it furnishes information regarding event occurrences and rule execution. The first part of this file contains event object names with their object ids and rule objects and rule ids. This information is generated when the event and rule objects are created at runtime.

The visualization tool parser reads the static information and stores the event and rule information in the event repository and rule repository respectively. The event and the rule repositories are linked list structures. With the help of this static information the user can see structure of an event/rule, which rule has subscribed to which event, individual instance and class level rules. The rule debugger requires the following information with respect to an event: name, classname with which the event is associated if it is a class level event, type (primitive or composite), signature of the method on which the event is defined

along with the event modifier whether it is raised before or after the invocation of the method and event expression if it is composite. The rule debugger requires the following information with respect to a rule: name, classname if it is class level, signature of the methods implementing the condition and action, context for which the rule is fired, coupling mode, trigger mode and priority.

The visualization tool takes in the above static and runtime information and parses it, then creates two tree-like data structures for events and transactions. In the event tree, primitive events are leaf nodes and composite events are seen as parent nodes of their components. The event tree grows from primitive events to the root, which is a composite event. The data structure which captures the nested execution of rules is an n-ary tree. The root node represents the top-level transaction of the application. When this transaction triggers a rule and since rules are executed as subtransactions, the child node of the top-level transaction represents the rule fired. This node in turn could trigger another rule and it is represented as the child node (subtransaction) and so on. The transaction tree grows in a top-down way: it starts from the top-level transaction and spans to the descendents. Presently the transaction tree nodes represent only rules. The events which cause these rules to fire are shown in a separate tree and are linked to the transaction nodes to indicate the firing of a rule by an event (primitive or composite).

5 Functionality of the Tool

The visualization tool's interface and functionality has been designed to provide as much information as possible in an uncluttered manner. Some information (e.g., event and rule detection and execution) is provided as part of the visualization whereas other information (e.g., objects held by a rule/subtransaction) is provided on a demand basis.

The user interface window consists of the following parts:

1. Predefined event information. All events defined in an application are arranged as a list inside a scrolled window. When an item is chosen by a mouse click, a dialog window pops up showing event type, name, method and when for primitive events and description of component events for composite events.
2. All predefined rules and their details including name, condition, action, priority, coupling mode, context, associated event are arranged in scrolled window in the same way as event information pane.
3. A group of push buttons provide various functions supported by the tool.
   HELP: on-line help.
   TRACE: run trace of rule execution.
   SELECT/ADD: select a subset of rules to monitor.
   CLEAR SELECT: delete the selection of rule subset.
   CONTMODE: select the continuous tracing mode.

STEPMODE: select the step tracing mode.
CLEAR GRAPH: clear the drawing-area windows.
QUIT: exit from the program.

4. Event/rule visualization window. During a trace, the upper half of this drawing-area window shows the event tree, with leaf nodes (primitive events) on the top. Composite events link with their component nodes through straight lines. Initially, since no event has been detected/raised, all nodes are in the color of grey.
   The lower half of the drawing-area displays rule execution tree, which demonstrates the nested nature of transactions. Each node stands for a subtransaction. Different colors are used for the three states of subtransactions: green for running, yellow for suspended and red for committed. Whenever a rule is fired a line connecting the transaction node of current rule and the triggering event is shown, and the color of the triggering event is changed to brown. The program can detect the type of display device automatically and use appropriate colors to represent different states of subtransactions. When monochrome display is used, an empty box drawn with dashed lines would represent a suspended subtransaction node, while a box with solid lines stands for a running node and a box filled in black corresponds to a committed subtransaction. This is demonstrated in the screen dumps at the end of this paper.
5. Execution console window. A drawing-area (extreme right) displaying execution information in plain text. The information includes all transaction/subtransactions with their ID, events that are raised and rules that are fired. This window records the event detection, subtransaction and rule firing sequence. This window can be scrolled to look the textual trace of execution. Colors differentiate various items displayed to make it more readable.

6 Visualization Using the Tool

The tool currently works as a post-execution visualization tool. In other words, after the execution of the application (successfully or otherwise), this tool can be invoked for observing the execution of rules, transactions, and subtransactions. The tool uses the information collected both at the pre-processing of the application and the actual execution of the application to graphically display primitive events, composite events, their detection, rules, and their execution as sub-transactions. Additional information on applications objects (e.g., events, rules, and database objects) can be obtained by an explicit action by the user. Below, we show a sequence of displays (or screen dumps) highlighting on the information provided to the user.

Figure 3 shows the output of a step-mode trace before any event is raised. It shows a display consisting of all the primitive and composite events specified (the event graph with event expressions shown graphically) in the application.

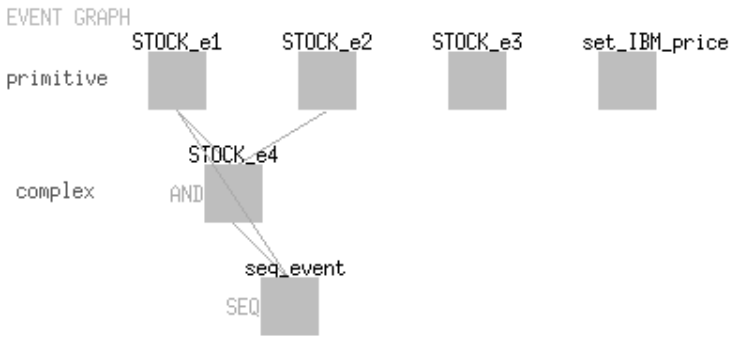

Fig. 3. Starting display of the Visualization Tool

In the event graph, there are 4 primitive events and 2 complex events. Lines connecting events show the composition of a composite event. It also shows that a top level transaction 1 and a subtransaction (not a rule) 100 has started. The colors of the event nodes indicate that no event has been detected.

Figure 4 shows the state of the same application when event stock_e4 is raised and rule R1 is triggered. It also shows the dialog when button R1 (for the rule R1 that has been fired) is pressed. The text pane contains a description on the triggered rule.

Figure 5 shows the display after the trace is completed. In the event graph, nodes that are shown in black represent events that were raised during the trace. In the rule graph all transaction nodes are filled in black because they are all committed. Lines connecting pairs of event/rule nodes indicate the relationship between triggering events and fired rules.

The above displays have been drawn from a simple stock market application consisting of a few events (both primitive and composite), rules, and subtransactions for exercising the functionality of the visualization tool.

7 Conclusions

In this paper, we have presented an overall architecture of a visualization tool especially tuned for the requirements of an active database system. We have also detailed the design and implementation of the visualization tool. The debugger currently supports visualization of rules as post-execution analysis. We have adopted an problem oriented approach to debugging of rules. In addition to merely showing the trace of rules the debugger also furnishes the context (events) in which the rules are fired. Additional information on most of the objects of interest (event, rule, transaction) can be obtained by the user.

Currently the visualization tool supports only post-execution analysis of rule

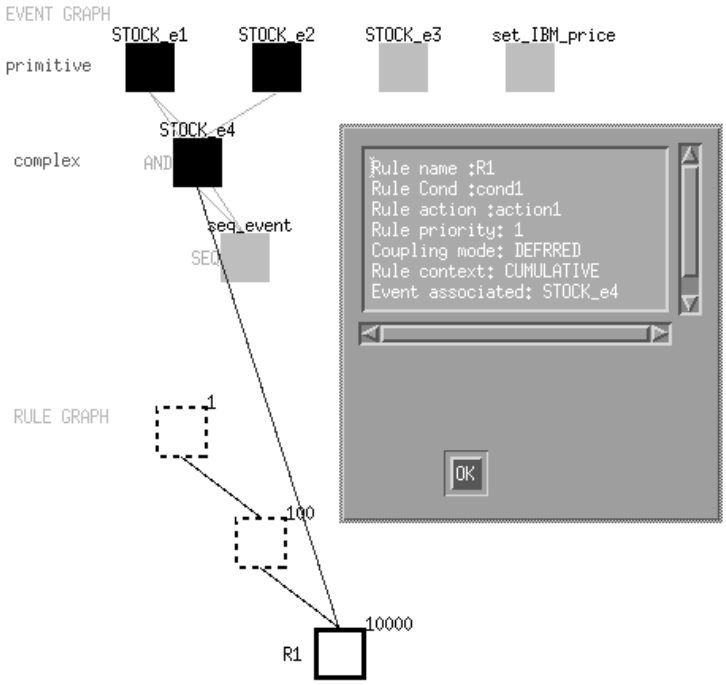

Fig. 4. Display of the Visualization Tool after Stock_e4 is raised

execution. Our next step is to support runtime visualization as well. We also plan on incorporating static analysis tools as part of the visualization toolkit so that runtime execution can be compared with static analysis. Both static analyzers for checking the termination, confluence, and observable determinism characteristics [AWH92] and debuggers (or visualization tool) for observing the run time behavior are extremely useful. In addition to rule execution, the composite event detection in various parameter contexts will also require feedback for ascertaining the correctness of parameter context used. Ideally (i.e., for the long term), it would be useful to have a visualization tool to which you can *specify* your expected behavior and the tool provides a visual feedback on how the actual execution differs from the specification and offers guidance for correction.

Thus far we have seen that all rules and events produced are shown by the visualization tool. This can become quite inconvenient for the user if he/she is dealing with large number of rules. Potentially we can identify two ways of pruning the tree: the user can either choose the rules or events he wants to monitor. This is possible in our case since events and rules are objects and more over there is a one to one correspondence between rules and transactions. When the tree is pruned in this way the child node may not be an immediate subtransaction of the parent node, but rather a descendant of the parent transaction.

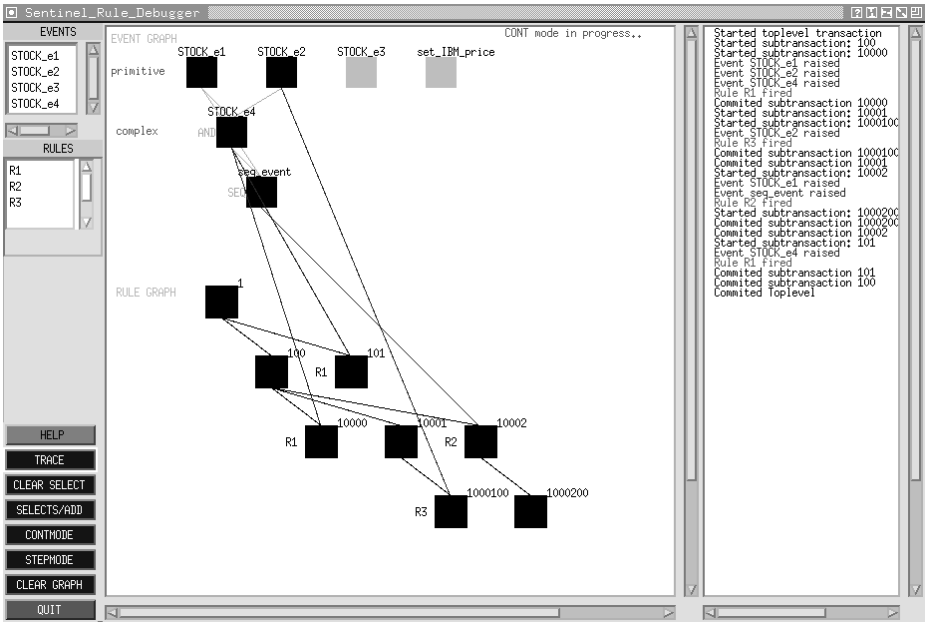

Fig. 5. Display after rule execution finishes

Interactive debugging is another feature that we plan on adding in the next version. The user should be able to selectively enable and disable a subset of events and rules for debugging purposes (in the run time debugging mode). This entails some changes to the Sentinel architecture to be able to stop and take into account user requests at run time.

Visualization tool is a first step towards the use of reactive capability for simulation applications. In simulation applications, the mechanism used typically for debugging is itself the front-end (or user interface). The availability of customizable visualization tool will help use the system for simulation applications, such as threat analysis, slow-motion execution of various situations.

References

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proceedings, International Conference on Management of Data*, pages 59–68, May 1992.

[Bad93] R. Badani. Nested Transactions for Concurrent Execution of Rules: Design and Implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, October 1993.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606–617, August 1994.

[CKTB95] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. In *Proceedings, International Conference on Data Engineering*, Feb. 1995.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, October 1994.

[DJP93] O. Diaz, A. Jaime, and N. W. Paton. Dear: A debugger for active rules in an object-oriented context. In *Proc. of the 1st International Conference on Rules in Database Systems*, September 1993.

[OOD93] OODB. Open OODB Toolkit, Release 0.2 (Alpha) Document. Texas Instruments, Dallas, September 1993.

[Tam94] Z. Tamizuddin. Rule Execution and Visualization in Active OODBMS. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, May 1994.

[WBT92] D. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–81, October 1992.

This article was processed using the LaTeX macro package with LLNCS style